Impact Analysis and Its Application in the Year2000 Problem.

A Thesis Submitted
in Partial Fulfillment of the Requirements
for the Degree of
Master of Technology

by
Himadri Sekhar Paul

to the

Department of Computer Science & Engineering
Indian Institute of Technology, Kanpur
May, 1998

Entered in system

CSE-1998-M-PAU-IMP

Certificate

Certified that the work contained in the thesis entitled "Impact Analysis and Its Application in the Year2000 Problem.", by Mr.Himadri Sekhar Paul, has been carried out under my supervision and that this work has not been submitted elsewhere for a degree.

[illegible]

(Dr. Sanjeev Kumar Aggarwal)
Associate Professor,
Department of Computer Science & Engineering,
Indian Institute of Technology,
Kanpur.

May, 1998

Abstract

The Year2000 Problem arises in systems which use two-digits to store year information in the date field and do not explicitly store the century information. The century is implicit and assumed to be '19'. These compact date-formats work well from year 1900 to year 1999, because most of the programs work under the same assumption. However these systems will fail to operate correctly in year 2000 and beyond.

The solutions proposed to solve this problem require change in variable declarations, that store date-value and sometime the parts of the code which handle such variables. It is a widely accepted fact that a complete automated solution to the problem is not possible. At the same time it is also true that the programmers alone cannot do the required changes. Therefore any tool that helps even in partial automation is a welcome step.

In this thesis a technique called *impact analysis* has been proposed to identify all the date-related variables. The technique is based on the *reaching definition analysis* and *du-chain* construction technique commonly used in the optimization phase of compilers. A tool which implements the technique has also been developed.

Acknowledgement

At the very outset I express my sincere gratitude **Prof. Sanjeev Kumar Aggarwal** for granting me work under his supervision. In spite of his various involvements with the department and Compute Center, IIT Kanpur he was always helpful and willing to share my difficulties. I am indebted to him for his kindness.

I am thankful to **Infosys Technologies Limited**, who supported this work. I am specially thankful to all the people in Infosys Y2K group for their support. I take this opportunity to thank **Prof. Pankaj Jalote** for his kind help and encouragement.

I am as ever grateful to my parents and my little sister, for bearing with me during this period. I am grateful to all of my class-mates who made my stay at this institute a joyous one. Their love and help has been a constant source of inspiration for me. Last but not the least my gratitude goes to my 'hall-IV D-mid' friends, Samarjit, Goutam-da, Jal-da, Sunu-da, Porel-da, Paddy-da, Sourav-da, Dhruba-da, Tanmoy-da, Gaya-da, Jyoti-da, Pila and Jay and also to Kola, Pradip-da, Sishir and Koushik for their affection.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 The Year2000 Problem

The Year2000 problem is an interesting phenomenon from both technical and sociological point of view. At the technical level, it concerns with the widespread practice of storing year in dates in two-digits format. For example, year 1998 is represented as '98' only. As a result at the end of 20th century, many software applications will stop working or produce erroneous results. At the sociological level, it is interesting to see how individuals and organizations react to the crisis. However, we will only concern ourselves with the technical aspect of the problem.

The problem leading to Year2000 challenge is really very easy to grasp. Yet, its consequences are very grave. Most programs and databases use two-digits and do not store the century information. For example, the Gregorian date format CCYY/MM/DD is represented as YY/MM/DD format and the Julian date format DDD/CCYY is represented as DDD/YY format. The century is assumed to be '19' and not explicitly mentioned. The problem is mainly in legacy systems, where memory is considered to be very precious and programmers made such an assumption in order to save storing space. In some systems the assumption was made just to save a few key-strokes by the data entry persons. This compact date format works well from year 1900 to year 1999 because most programs operate on the same assumption; but it will fail for year 2000 and beyond. This problem is synonymously referred to as Y2K problem, Century Date Conversion problem(CDC problem), Millennium problem etc.

1.2 The Year2000 Problem Exposure

The software systems, where the year detail is represented by just two-digits, are exposed to the Year2000 problem. Hardwares that do not support four-digits year field format will also be affected. For example, PC-BIOS stores year field in CMOS-ROM in two-digits format. There are bar-code systems which use two-digits year format as part of their schemas. The Year2000 problem will also affect those systems which read and use these bar-codes. In our world where the systems are highly interconnected through network; it has been predicted that systems, which although do not have Year2000 problem in its own environment, will be exposed to the problem by other Year2000 problem affected systems on the network in the form of data transfer [8, 15].

The Year2000 problem is further compounded with numerous variations of date representation and mathematical calculations done on these dates. The following section gives few of the classifications of the Year2000 problem exposure.

1.2.1 The Year2000 Exposure Classification

- **Incorrect century**
  The systems, which use two-digits year-field, assume the century field to be '19'. These systems ignore the century field during date entry or update. Hard copy output of these systems put '19' in the century field by default.

- **Incorrect field format**
  Date formats such as YY/MM/DD(Gregorian) or DDD/YY(Julian) bind the system to operate within a fixed 100-years window ranging from year 1900 to 1999.

- **Arithmetic Calculation**
  Arithmetic with the two-digits year representation cannot work outside the year range 1900-1999. It will produce anomalous results on and beyond year

2000. For example, difference between year 2010 (represented as '10') and year 1990 (represented as '90') is calculated as (10 - 90) = -80 years, where the correct result is (2010 - 1990) = 20 years.

- **Leap year calculation**
  Year 2000 is a leap year and is represented as '00'. Since the default century value is '19', computers will see year 2000 as year 1900, which is not a leap year. Potential exposures caused by the identification of year 2000 as non-leap-year are [6]:

  - Day-in-year calculation. The year 2000 has 366 days and not 365 days.
  - Day-of-the-week calculation. February 28, 2000 is a Monday and March 1, 2000 is a Wednesday. Systems will fail to calculate correctly the day-of-the-week. For example, the ADA language relate database exhibit strange Year2000 problem [19]. It declares year 2000 as leap year and seems to calculate correctly the day-of-the-week February 29, 2000, but fails to calculate correctly that of March 01, 2000.

- **Data Integrity**
  The Year2000 problem affected systems will see year 2000 as year 1900 and will fail to distinguish between events occurring in the year 1900 and 2000. Events occurring in the year 2010 will appear to have already occurred in the year 1910. Time will appear to have reversed.

- **Sequence**
  Sorting with date as the key value will produce erroneous result. Records of year 2000(represented as '00') will appear before year 1999(represented as '99') in sorted sequence.

- **Year value with special meaning**
  In some systems some year values are assigned with special meaning and the meaning is hard-coded in the code. For example, systems may treat '99' in year field as 'date value not available', or '00' in year field may signify 'this record has been expired', etc.

- **The Century Rollover**
  The century rollover means the transition of a system from one century to the next. For example, century rollover for 21st century means transition of the system-date from 31st December, 1999 to 1st January, 2000. Systems which are affected by Year2000 problem very often fail in 21st century rollover and they are said to be showing century rollover syndrome.

Leon A. Kappleman has coined the term DRAGON to represent the three possible outcomes of the Year2000 problem [14]. DRAGON is Date Related Abend, Garbage Or Nothing. The term 'Date Related' is used because of the nature of the problem. The problem has three consequences on the systems.

i) Abend, which in computer linguistic means total shutdown of the system. This is the extreme impact that can occur to the system.

ii) The system will produce erroneous outputs *i.e.* Garbage.

iii) In some systems the problem will do Nothing.

1.3 Solutions and Techniques

All the solutions proposed to solve the Year2000 problem require changes in the program and/or databases. The process of incorporating the required changes must be systematic and is commonly referred to as *Year2000 conversion* process. The solutions proposed for the problem can be classified into three categories, namely Data approach, Procedural approach and Encoding/Compression approach. All the three approaches along with their advantages and disadvantages are discussed below [1, 8, 9, 10].

1.3.1 Data approach

The data approach involves expansion of the date fields from two-digits year format (YY) to four-digits year format (CCYY) to include century information, both in source code as well as in stored data. The solution requires changes in all the data files and databases and also the application programs that refer to or use the changed databases.

Advantages

- This is the ideal solution as the converted application will survive upto the year 9999.
- This approach requires easier code upgradation effort and results in simpler date logic in programs, i.e. conversion effort is minimized as most changes will be confined to the data declaration part.
- This conversion approach is both consistent and accurate. It eliminates two-digits year ambiguity and Year2000 rollover issue simultaneously. Moreover, this approach ensures consistency with newly developed century-compliant systems.

Disadvantages

- This method requires conversion of virtually all programs of the system.
- Since the conversion of the programs and the data must occur simultaneously this approach requires very careful project management. The process may force shutdown of the system until the whole conversion process is over.
- Data conversion process in this approach may be costly. Archived data may also need conversion or support by special logic. The process may become complex, potentially needing support for the new and old date formats simultaneously.

1.3.2 Procedural approach

The procedural approach is based on the observation that two-digits year field naturally offers us a 100-years window. Since these systems assume century to be '19', two-digits year field gives them a fixed window ranging from year 1900 to 1999, in which these systems work properly. The base year in this case is 1900. This approach involves changes only in the source codes to incorporate proper logic such that any value can be imposed as the base year and the century of the two-digits year be correctly interpreted. This solution does not require expansion of the year field neither in data nor in the declaration part of the code.

There are two variations of this scheme.

- Fixed Window Technique
- Sliding Window Technique

Fixed Window Technique

The fixed window technique uses a static 100-years interval that generally spans over century boundary. For example the window can be 1964-2063, where the base year is 1964. The application system can be modified to infer the century of the two-digits year based on the following logic:

```
if year >= 64 then century of the year is 19.
if year <= 63 then century of the year is 20.
```

Sliding Window Technique

In this case the 100-years window is dynamic. The technique uses a self-advancing 100 years interval that generally crosses the century boundary. The user specifies the number of years in the past and the number of years in the future relative to the system date(generally the current year). The system maintains the 100-years window for the data. The main advantage of the approach is that the window is automatically advanced without any programming change.

This technique is suitable for applications which process data of a predefined interval. The sliding window technique can follow the following logic to infer century of the year. The example assumes past year number is 60 and the future year number is 40.

```
In 1996,  If year >= 36 then century of the year is 19.
          If year <= 35 then century of the year is 20.
```

```
In 1997,  If year >= 37 then century of the year is 19.
          If year <= 36 then century of the year is 20.
```

Advantages

- No data conversion required in this technique. Existing data and archive data remain unexpanded.
- Programs can be changed and tested in isolation from other interdependent systems.
- The exposure to change is limited to the parts of the program where the logic is modified.

Disadvantages

- Some programs may require extensive code changes, because they may require extra logic to cater for century rollover and also because the century inference process can be complex.
- The solution fails in systems which operate within a span of 100 years or more.
- The solution may require data-bridge in case of date-exchange with systems which use four-digits year field.
- Sorting complexity may increase in this approach.

1.3.3 Encoding/Compression approach

The previous two solutions neglects the fundamental cause of the Year2000 problem, *i.e.* storage overflow. Computers can store much more information in current date-formats (YYMMDD or YYDDD) than they are allowed to do. The Encoding/Compression method use the existing storage space for date to store more data in the form of more concise date-representation. The main advantage of the schema is that in doing so, we get a much larger window than 100-years. There are many encoding/compression techniques available [1, 18]. Some examples are given below:

- The CYYDDD Format : The existing YYMMDD is converted to CYYDDD format. This scheme can give an year-window of 1000 years long.

- The DDDDDD Format : In this case YYMMDD format is replaced by DDDDDD format. This representation can store 1 million days. The stored value is the number days elapsed since some base date, say January 1, 1900. Then it can track dates past year 4600.

- Conversion of the number representation schema from decimal to hexadecimal. Two hexadecimal-digits can provide an year-window of 255 years.

- Conversion of the year data type from the two-bytes character representation of the two-digits year to a one-byte unsigned packed decimal representation; and using the free byte to store two-digits century field as unsigned packed decimal.

The solution requires changes in both the data and the programs. It also requires simultaneous conversion of data and all the applications which refer to or use the converted data.

Advantages

- This approach does not require expansion of two-digits year format to four-digits year format, yet it gives the system a larger year-window to operate in comparison to the 'procedural approach'.

Disadvantages

- The approach requires simultaneous changes in all the program units and data.

- Although the schema does not require expansion of date fields, it requires updating of data. At the same time it requires extra level of processing in all the places in the code where the encoded year field is being used.

1.4 Problem Definition

All the approaches to the solution of the Year2000 problem require identification and modification of date-variables as well as those parts of the program which handle those date-variables. Modification in any part of the program has a rippling effect throughout the program, i.e. such changes affect other parts in the program including other variables. For example, consider the following piece of code:

```
01 JOIN_DATE              DATE  pic 9(6).
01 A                      INT   pic 9(6).
01 B                      INT   pic 9(6).
.
.
.
MOVE JOIN_DATE TO A.
.
.
.
MOVE A TO B.
```

Now assume that the date-variable `JOIN_DATE` has been expanded to eight-digits. Since the variable `A` can hold only six-digits of information, two-digits of information will be lost at the `MOVE` statement. This implies that in order to retain the complete information, variable `A` should also be expanded to eight-digits. In this case variable `A` is under the direct *impact* of the date-variable `JOIN_DATE`. Similar argument can be applied to variable `B` also. Here `B` is under the direct impact of the variable `A` and under the indirect impact of the variable `JOIN_DATE`.

Form the above example it is clear that changes are required for both date-variables and non-date-variables, which interact directly or indirectly with the date-variables. Therefore we require to identify all the variables those are likely to obtain year value and also those part of the code which are likely to manipulate on such value. The process of identification of such variables is called *Impact Analysis*.

As argued by Peter de Jager in his article *Biting the Silver Bullet* [7] no fully automated solution to the problem, which he refers to as *silver bullet*, is possible. He has identified more than a dozen of challenges a sliver bullet has to overcome, and it is unlikely that a single tool can overcome all of them. At the same time it is also apparent that only human intervention for the conversion process is not practical. Therefore tools those help in Year2000 conversion process are required.

1.5 Existing Tools

The Year2000 problem has captured a world-wide attention, accompanied with bizarre predictions. Many existing companies started addressing the problem and many service companies were established with their only attention to the problem. As a result numerous tools have come up. They address various aspects of the conversion process of the Year2000 problem. Nicholas Zvegintzov, in his article *A Resource Guide to Year 2000 Problem* [22], has classified the specialized Y2K tools along six perspective:

1. *Inventory Analysis.* Tools that support the task of identifying the executable software inventory.

2. *Recovering source from object.* Tools for analyzing code for which the source cannot be reliably identified.

3. *Project estimation.* Tools for estimating the work load of Y2K conversion process.

4. *Analysis and conversion.* Tools for finding and changing code or data structures for Y2K conversion.

5. *Time simulators.* Testing tools which simulate and modify clock time to provide testing environment.

6. *Date libraries.* Libraries for standard date formats and date calculations.

In this thesis we are concerned only with the tools for analysis and conversion. Analysis at the code level is very important for Year2000 conversion process. The Year2000 conversion process, as discussed earlier, requires identification and modification of date-variables. Analysis requires identification of system components and how they interact with data and among themselves. This makes the code modification a secondary job. There are many tool sets having specialization for date and time analysis. There are actually few tools which support direct conversion/change of code. Here are few example of tools of this kind [1, 3, 4, 21, 22].

- Analyzer 2000 is a simple inventory tool that scans file for date patterns, flags them and makes them editable. The tool works under MVS and OS/2 environment with Cobol, JCL and RPG as input language. The tool is from *Ironsoft Inc., Madison, WI.*

- CA-Impact/2000 is a tool with capability of trace, inventory and edit potential date-related fields. Limited impact analysis is done within one program unit. The tool is marketed by *Computer Associates International, NY.*

- Cayenne 2000 is an analysis tool from *Cayenne Software Inc.* The tool finds potential date fields and performs limited impact analysis.

- Century File Conversion locates and inventories date-related fields in source code. The tool is marketed by *Quintic Systems.*

1.6 Scope of the Thesis

As discussed earlier the emphasis of the Year2000 conversion is on the analysis of the programs. The objective of analysis of programs is the identification of date and non-date variables which may hold date-values during program execution. At the same time the analysis process reveals the places of the programs where such variables are being used. Obviously these places are the potential places to carry out changes for Year2000 conversion. We have proposed a technique called *Impact Analysis* for identification of variables which may hold date-values during program execution and also the places in the program where the variables are being used.

Impact analysis of a given variable traces the path of a data that can be associated with the given variable. If the variable given is a date-variable, then the technique identifies the path of a date-value, originating from the specified date-variable. An editing tool has also been implemented based on the impact analysis technique. Brief discussions on the following chapters are given below:

- Chapter 2 explains the impact analysis technique along with its theoretical background. It includes discussions on control-flow graph, reaching definition analysis and du-chain which lead to impact analysis.
- Chapter 3 discusses the functionalities and high level design for the Power Editor. It also discusses about the individual modules of the tool.
- Chapter 4 includes results, along with discussion of the environment in which the tool runs, code size and testing results.
- Chapter 5 draws the conclusion.

An user's manual for the tool is appended as appendix A at the end of this thesis. Few test results of the tool are included as appendix B.

Chapter 2

Impact Analysis Technique

2.1 Introduction

The impact analysis technique forms the back-bone of our tool. The technique is based on the data-flow analysis, which is a common method used for the code optimization in compiler domain [2, 11]. More specifically we are using the *live variable analysis* and *du-chain* construction methods as the basis of the technique. Discussions on the mentioned techniques assumes that the input program has been reduced to its equivalent intermediate representation. From this point onward we shall assume that the techniques are applied on input programs in 3-address intermediate code and we shall refer to each 3-address code as a *statement.*

2.2 Basic Block

A *basic block* is a sequence of consecutive statements in which flow-of-control enters at the beginning and leaves at the end, and no other flow-of-control branches in or out from the basic block.

The algorithm for identification of basic blocks from a program is given in the following page:

Algorithm : Partition of basic blocks.

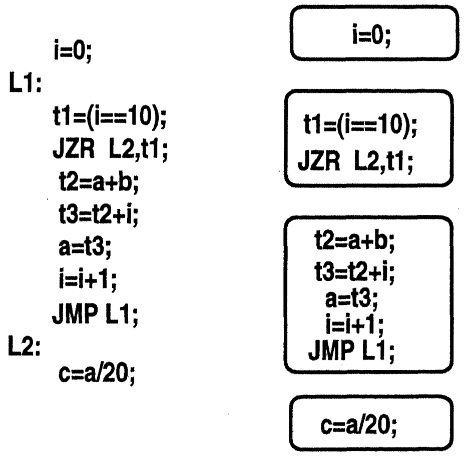
Input : A sequence of 3-address statements.

Output : A list of basic blocks with each 3-address statement in exactly one block.

Method :

1. We first determine the set of *leaders*, the first statements of basic blocks. The rules are as follows:
   i) The first statement is a leader.
   ii) Any statement that is the target of a conditional or an unconditional 'goto' statement is a leader.
   iii) Any statement that immediately follows a 'goto' or a conditional 'goto' statement is a leader.
2. For each leader, its basic block consists of the leader and all statements upto but not including the next leader or the end of the program.

Figure 1 shows the basic blocks identified using the above algorithm.



```
     i=0;
L1:
     t1=(i==10);
     JZR  L2,t1;
     t2=a+b;
     t3=t2+i;
     a=t3;
     i=i+1;
     JMP L1;
L2:
     c=a/20;
```

```
i=0;
```

```
t1=(i==10);
JZR  L2,t1;
```

```
t2=a+b;
t3=t2+i;
a=t3;
i=i+1;
JMP L1;
```

```
c=a/20;
```

Figure 1: Basic Blocks.

2.3 Control-Flow Graph

Flow-of-control in a program means the possible execution sequences of the program in run time. We can add the flow-of-control information to the set of basic blocks to get a *Control-Flow Graph.* The flow-of-control information forms the directed edges of the control-flow graph and the basic blocks are the nodes of the graph.

One node of the control-flow graph is distinguished as the *initial* node and the program execution starts from the 'initial' node. If there is a directed edge from the basic block B_1 to B_2, then the execution of the basic block B_2 can follow that of B_1 in some execution sequence. This can only happen when:

1. There is a conditional or an unconditional jump statement from the last statement of B_1 to the first statement of B_2.

2. B_2 immediately follows B_1 in order of program and there is no unconditional jump from the last statement of B_1.

Then B_1 is said to be the *predecessor* of B_2 and B_2 is the *successor* of B_1. Figure 2 shows the control-flow graph of a simple loop statement.

```
      i=0;
L1:
      t1=(i==10);
      JZR  L2,t1;
       t2=a+b;
       t3=t2+i;
       a=t3;
       i=i+1;
      JMP L1;
L2:
       c=a/20;
```

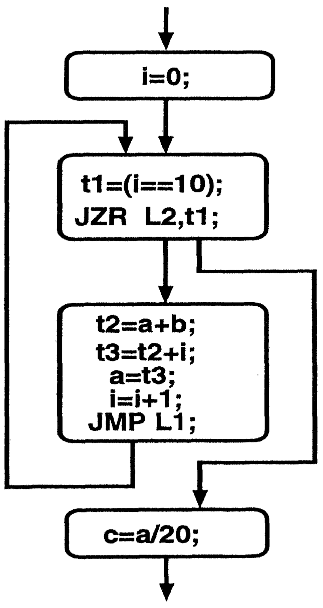


Figure 2: Control-Flow Graph.

2.4 Data-Flow Analysis

Data-flow analysis is the process of ascertaining and collecting information about the possible run-time modifications and usage of certain quantities in the program. Since data or 'values of variables' are the fundamental information that the program manipulates, usually the values of variables are taken as the quantities under scrutiny in data-flow analysis.

We distinguish among several levels of data-flow analysis:

(a) statement

(b) basic block (intrablock)

(c) procedure (intraprocedural) and

(d) program (interprocedural)

Some authors describe statement level and basic block level analysis as *local* data-flow analysis and intraprocedural and interprocedural analysis as *global* data-flow analysis.

Data-flow information is collected hierarchically. Statement level analysis involves analysis for a single statement. In this kind we calculate what variables have their values modified, preserved and used, if control proceeds through the statements. Statement level analysis is used in basic block level analysis. We examine each of the statements in a basic block in order and calculate similar information for the block. Basic block level information along with control-flow graph leads to information about the a procedure. And with similar information about procedures, we can construct information about the whole program in program level analysis. However, collection of intraprocedural information through control-flow is somewhat complicated. Even information about particular indivisual statement, such as 'call' statement, may not be immidiately available.

2.5 Global Data-Flow Analysis

Reaching definition and *live variable* analysis are the two typical applications of the information generated through global data-flow analysis. We shall discuss only reaching definition analysis as the basis of impact analysis which is easily obtained from the reaching definition analysis.

Here we recall a few definitions which will be used in further discussion.

2.5.1 Definitions

1. Point: A point in a basic block is the possition between two adjacent statements, as well as the position before the first statement of the block and the last statement of the block.
   If there are n statements $s_1, s_2, \ldots s_n$ in a basic block, there are $n+1$ points $p_1, p_2, \ldots p_{n+1}$ in the block, such that,

$$p_{i+1} \text{ is in between } s_i \text{ and } s_{i+1};\ \forall i : 1 \leq i \leq (n-1)$$
$$\text{and } p_1 \text{ is before } s_1 \text{ and } p_{n+1} \text{ is after } s_n.$$

   Figure 3 shows the points in a basic block.

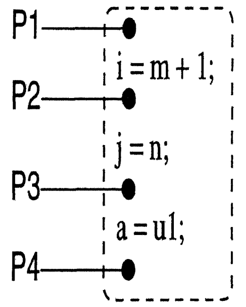


Figure 3: Points in a Basic Block.

2. Path in the Control-Flow Graph: A path from point p_1 to p_n is a sequence of points $p_1, p_2, \ldots p_{n+1}$ such that for each i, $1 \leq i \leq n$ either of the following two conditions holds:
   - p_i is the point immediately preceding a statement and p_{i+1} is the point immediately following that statement in the same block.
   - p_i is the last point of same block and p_{i+1} is the first point of a successor block.

Figure 4 shows a path in control-flow graph from point p_1 to p_{12} and the points it passes through can be given as : [p_1,p_2,p_3,p_4,p_5,p_6,p_7, p_8,p_9,p_{10},p_3,p_4,p_5,p_{11},p_{12}]

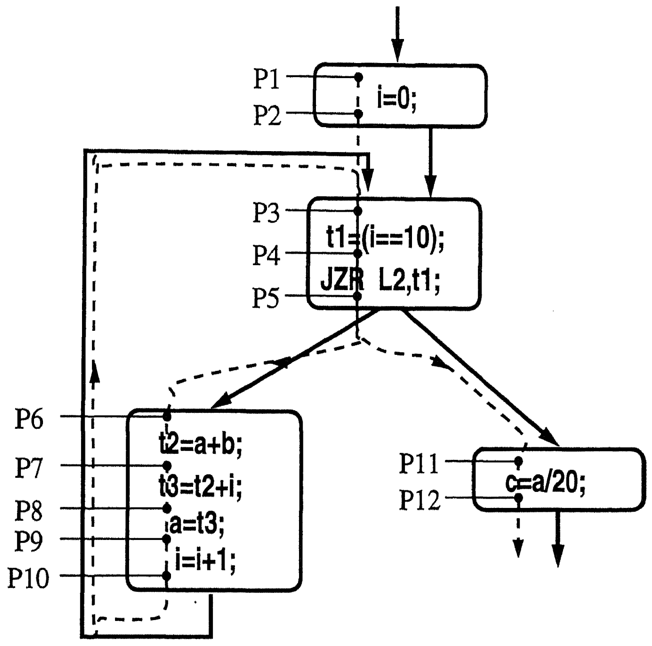


Figure 4: Path in a Control-Flow Graph.

3. **Definition of a variable:** A definition of a variable x is a statement that assigns a value to x.

4. **Use of a variable:** Use of a variable x is the statement where x occurs at the right hand side as an operand.

   Consider the following piece of code in 3-address representation.

```
1.      i = 0;
2.      k = 10;
3.      x = 5.0;
4.      b = 2.25;
    L1:
5.      t1 = k - i;
6.      JNZ L2,t1;
7.      t2 = x + b;
```

```
8.        t3 = 50.0 - x;
9.        x = t3;
10.       t4 = i + 1;
11.       i = t4;
12.       JMP L1;
     L2:
          ⋮
```

Here the definitions of the variable i are at statement nos 1 and 11 and uses of i are at statement nos 5 and 10. Similarly, definitions of variable x are at statements 3 and 9 and its uses are at statement nos 7 and 8.

2.5.2 Global Data-Flow Equations

Information about data or value of variables, which are ascertained from data-flow analysis are represented as set a of variables. Typically, there are 4 kinds of sets associated with every node of a control-flow graph.

- in[B] is the set of definitions that enters the basic block B.
- out[B] is the set of definitions that comes out of basic block B.
- gen[B] is the set of definitions generated within the basic block B. These are actually locally exposed or locally generated definitions for node B. This is the set of definitions created by B as far as the outside world is concerned.
- Killing of a definition: A definition d of a variable x is said to be 'killed' at point p along a path starting from the point immidiately following d to the point p, if along that path there is another definition of x. The definition d is *live* at p if d is not killed along any path from the point immidiately following d to p.
- kill[B] is a set of definitions being killed in the basic block B.

2.6 Reaching Definition Analysis

We say a definition d reaches a point p if there is a path in control-flow graph from the point immidiately following d to p such that d is not 'killed' along that path.

2.6.1 Data-Flow Equations for Reaching Definition Analysis

1. For each basic block B:

$$in[\mathrm{B}] = \bigcup_{x \in PRED[\mathrm{B}]} out[x]$$

If $PRED[\mathrm{s}] = \Phi$ then $in[\mathrm{s}]{=}\Phi$.

2. For each basic block B,

$$out[\mathrm{B}] = gen[\mathrm{B}] \cup (in[\mathrm{B}] - kill[\mathrm{B}])$$

3. Combining 1 and 2,

$$out[\mathrm{B}] = gen[\mathrm{B}] \cup ((\bigcup_{x \in PRED[\mathrm{B}]} out[x]) - kill[\mathrm{B}])$$

If $PRED[\mathrm{s}] = \Phi$ then $in[\mathrm{s}]{=}\Phi$.

- In the above equations $PRED[\mathrm{B}]$ denotes the set of *predecessor* blocks of basic block B.

Since equations 1 and 2 do not necessarily have a unique solution for *out*, we desire the smallest solution.

2.7 du-Chain

The construction of *du-chain* or *definition-use chain*, for a definition d of a variable x, involves identification of all the uses of x, where *d reaches*. For a use of the variable x in statement s, we calculate whether the definition d reaches from d to s along any path. If so then s is a part of the du-chain of d. Definition-use chain of definition d includes d as the head of the chain and followed by all the uses of x, where d reaches.

2.8 Impact Analysis

A definition d of a variable x signifies a piece of data being associated with variable x. The du-chain of a definition d signifies the association period of the data/value with x. In the current application we want to trace the complete life-period of a data which may be associated with more than one variable. A data value can be replicated to other variable through the following:

(a) Assignment to other variables and

(b) Used as argument of a 'call' to function.

An assignment statement repliacates and associates data with the variable to which the assignment is done. Similarly the formal argument of a 'call' statement replicates the value to the dummy argument of the function being called. The impact analysis technique tries to capture the complete flow of a data/value throughout the program.

Impact analysis technique is essentially based on the reaching definition analysis technique and the information generated by it is stored as an impact chain, which is an extension of the du-chain.

2.8.1 Definitions

- Seed Variable is a variable for which impact analysis is to be done.
- Seed Definitions are the defintions involving the *seed variable*.

2.8.2 Impact Analysis and Impact Chain

The concept of impact analysis technique of a seed definition is recurrsive. At the basic step we do the reaching definition analysis of the seed defintion. As a result we obtain a du-chain of the seed definition. Impact analysis is done on the definitions in the du-chain. The definition of *impact chain* of a seed definition d is consequently recurrsive and is given as follows:

1. Definition d is the head of the impact chain.
2. The 'uses' of d are part of the impact chain.
   (a) d' is a definition which uses d in the chain.
   (b) impact chain of d' is also in the chain.

The impact chain of a definition may be a chain larger than the corresponding du-chain. For example, consider the following piece of code, where we want to find the impact chain of the variable a.

```
01  :  int i,a,b;                          /* Declarations of i,a and b */

02  :  a=0;                                          /* Definition of a */
03  :  b=10;                                         /* Definition of b */
              :
07  :  for(i=a;i<=20;i++){                                /* Use of a and
                                             definition and use of i */
              :
13  :     }
14  :  b=a+10;                                              /* Use of x */
15  :  printf("Enter a value:");
16  :  scanf("%d",&x);                          /* Next definition of x */
17  :  if(a>b) {                                        /* Use of a and b */
              :
21  :     }
```

The du-chain and the impact chain of the definition of the variable 'a' at line no. 2 is shown in the figure 5.

▪ *Impact Analysis Algorithm*

The algorithm for impact analysis, based on the reaching definition analysis is given in the following page:

Algorithm : Impact analysis, based on reaching definition analysis.
Input : The control-flow graph of the program, the seed variable x.
Output : Impact chains corresponding to the seed variable.
Method :

```
Step 1:

    Find syntactically all the occurrences of the
    definition of the seed variable $x$. Call them
    *seed definitions.*

Step 2:

    For each 'Seed Definition' $s$ do
    {
        $R$  =  { $I$ | $I$ is a definition and
                the seed definition $s$ *reaches* $I$
                and $I$ uses $x$ };
        $P$  =  $\Phi$;
        For each definition $d$ in $(R - P)$ do
        {
          $v$   =  variable defined in $d$;
          $R'$  =  { $I$ | $I$ is a definition and
                   $d$ *reaches* $I$ and $I$ uses $v$ };
          $R$   =  $R \bigcup R'$;
          $P$   =  $P \cup \{d\}$;
        }
    output $R$;
    }
```

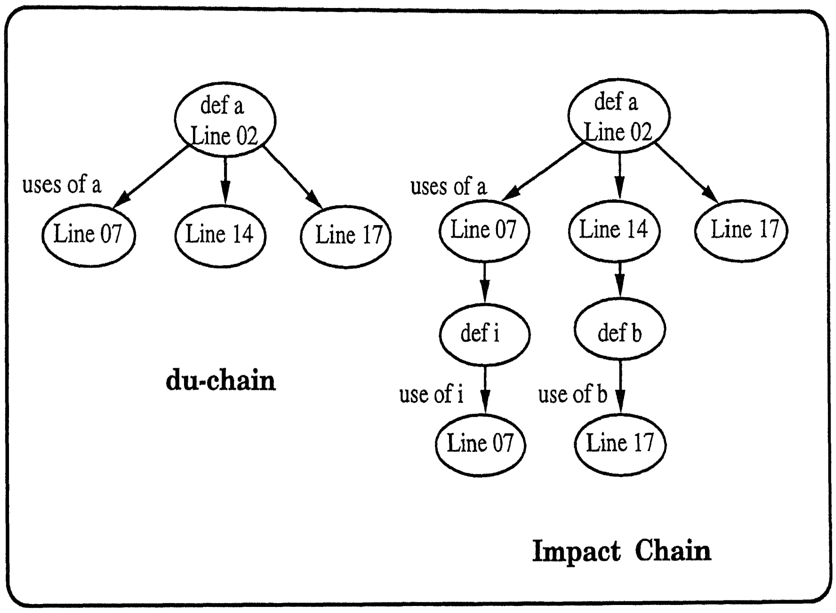


Figure 5: du-chain and impact chain of variable ‘a’.

Proof of Convergence:

Proof: The proof of convergence of the impact analysis algorithm uses the convergence of reaching definition algorithm.

It is easy to see that step 1 of the algorithm converges. Step 1 parses the program once and collects all the definitions of x where x syntactically matches the seed variable. The number definitions it collects is at most equal to the number of definitions in the program and since the number definitions in a program is finite, $|R|$ is finite. Therefore, step 1 runs for a finite amount of time.

R is the set of seed definitions collected in step 1 and $|R|$ is finite. P is the set of definitions for which du-chain construction has been done. Initially P is empty. In each iteration of the ‘for’ loop in step 2, the algorithm computes the du-chain for a definition d, where d is an element of $(R - P)$. $(R - P)$ is the set of definitions for which du-chain construction has not yet been done. Within the ‘for’ loop computation of R' uses the reaching definition analysis algorithm. Since the reaching definition analysis algorithm takes a finite time; each iteration of the loop also takes a finite mount of time. It is to be observed that both $|R|$ and $|P|$ never decreases. In each iteration $|P|$ increases by 1. Since R is a set of definitions and at most it can be the set of all definitions in the program, which is finite. Eventually, $|R-P|$ will be 0 and the iteration will terminate. Therefore, the number of iterations of the loop is finite.

This proves that the algorithm converges. ∎

∎ *Few Definitions*

- Impact Variable: Variable x is said to be an *Impact Variable*, if a definition of x is in the impact chain.

- Impact Points: All the definitions and the uses in a impact chain are called the *Impact Points*.

Chapter 3

Design Approach of a Power-Editor based on Impact Analysis Technique

3.1 Functionalities of Power-Editor

We have developed an impact analyzing tool which runs at the back-end of an editor. The tool is interfaced with the editor. The interface allows the users to use the impact analyzing tool along with the normal editing facilities of the editor. The editor and the impact analyzing tool is packaged together and is called the *Power-Editor*. The Power-Editor has the following functionalities:

1. All the interactions between the impact analyzing tool and the user is done through the editor.

2. The user can invoke the impact analyzer from the editor. When invoked the tool asks for the seed variable, does impact analysis on the program in the current buffer of the editor and produces the impact chains for the specified seed variable.

3. The user can browse through the impact chain with simple key-strokes and the editor takes the cursor to the corresponding positions in the program. The user can edit the program simultaneously.

3.2 Implementation Strategy

The impact analyzing tool has two main modules, the *Impact Analysis Engine*(IAE) and the *Impact Chain Processor*(ICP). The IAE has two submodules, namely *Intermediate Representation Generator*(IRG) and *Impact Analyzer.*

Figure 6 shows the modules and their interfaces.

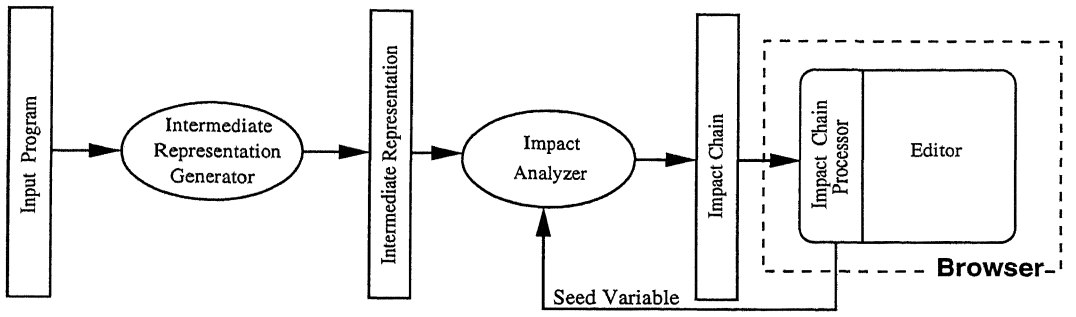


Figure 6: Modular design of the tool.

3.3 Discussion on the Design Modules

3.3.1 Intermediate Representation Generator(IRG)

This module produces an equivalent intermediate representation of the input program in 3-address intermediate code. IRG uses the front ends of compiler *i.e.* the lexical analyzer and the parser. There is a specified format in which the intermediate code is stored in a temporary file and it is used in the subsequent phases of the tool. The IRG expects a correctly written program as input and produces the corresponding 3-address intermediate code as output.

3.3.2 Impact Analyzer

This is the core module and performs impact analysis on the input code which is in 3-address intermediate representation. The input to this module is produced by IRG. The impact analyzer takes a seed variable from the user and produces impact chains for all the seed definitions. The impact of the seed variable can propagate from the main program unit to other functions, possibly to some other files (*e.g.* `included` files) through function calls. The impact analyzer is able to follow such data flows.

The impact analyzer performs the followings in order:

1. Generates the basic blocks from the input.

2. Adds control flow information to the basic blocks and produces the control-flow graph.

3. Takes a seed variable and finds out all the seed definitions. Then runs impact analysis algorithm and produces the impact chains corresponding to the seed definitions.

■ *Implementation*

The implementation has a small deviation from the impact analysis algorithm. Consider, some impacted variable is used as an argument in a 'call' statement. Let d_a be definition of the dummy parameter of the called function to which the argument value to be assigned. Obviously d_a is impacted through the 'call' statement and we can obtain an impact chain for d_a. The impact analysis algorithm implies that the impact chain of d_a is a part of the main impact chain. But the implementation stores it as a separate impact chain. This deviation although increases the no of impact chains, it reduces computation. But the total number of distinct impact points in both the cases remains the same. The implementation restricts each impact chain within one function unit.

Input : The control flow graph of the program. The seed variable 'x'.
Output : 'Impact Chain'.
Method :

Define the following sets :

Declarations :

D_s	=	Set of definitions which are the seed definitions for impact analysis.
$Instr_s$	=	The 'Seed Definition' which is currently under scrutiny.
$SeedVar$	=	Variable defined in $Instr_s$.
M	=	Set of statements which are already included in the impact chain corresponding to the current seed definition.
I_s	=	Set of statements which are impacted by $Instr_s$.
I_{call}	=	Set of definitions which are impacted through a 'call' to a function by the current seed variable.
IQ_s	=	Queue of seed definitions.
IQ_i	=	Queue of statements impacted.

Algorithm :

```
$D_s = \{ I \mid I$ is an statement and $y$ is $defined$ in $I$ and
        $y$ syntactically matches the seed variable $x$ $\}$;

For all element $e$ in $D_s$, enqueue $e$ in $IQ_s$;

while ($IQ_s$ not empty) do

{

    $Instr_s$ = dequeue from $IQ_s$;

    $SeedVar$ = variable $defined$ in $Instr_s$;

    $M = \phi$;

    $S = \{ I \mid I$ is an statement and $Instr_s$ $reaches$
            $I$ and $I$ is a $use$ of $SeedVar\}$;
    $I_{call}$ = { $I$ | $I$ is the first statement of the header basic
                  block of function ‘func’ and the definition $Instr_s$
                  is used to ‘call’ function ‘func’};
    $M = M \bigcup I_{call}$;
    $I_{call} = I_{call} - D_s$;
    for each statement $e$ in $I_{call}$ enqueue $e$ in $IQ_s$;
    $D_s = D_s \bigcup I_{call}$;
    $M = M \bigcup Instr_s$;
    $M = M \bigcup S$;
    for all element $e$ in $S$, enqueue $e$ in $IQ_i$;
    while ($IQ_i$ not empty) do
    {
        $I_i$ = Dequeue from $IQ_i$;
        $SeedVar$ = variable defined in $I_i$;
        $S$ = { $I$ | $I$ is an statement and $I_i$ reaches I
              and $y$ is $defined$ in $I$ and
              $SeedVar$ is $used$ in $L$ };
        $N = S - M$;
        Enqueue each element of $N$ in $IQ_i$;
        $M = M \bigcup S$;
        $I_{call}$ = { $I$ | $Instr_s$ is used to ‘call’ a function $f$
                  and $Instr_s$ is the formal argument for some
                  parameter $x$ of $f$ and $x$ is declared at $I$};
        $M = M \bigcup I_{call}$;
        $I_{call} = I_{call} - D_s$;
        for each statement $e$ in $I_{call}$ enqueue $e$ in $IQ_s$;
        $D_s = D_s \bigcup I_{call}$;
    }
}
```

3.3.3 Impact Chain Processor(ICP)

The ICP provides an interface with an editor. The ICP maintains the impact chains in its internal list data structure and provides an abstract view to the user. Each impact point in the impact chain is a position in some program file and the ICP takes the cursor to that point in the file when the user wants to see that point. So, while browsing through the impact chains the user only sees the cursor taken to the points in program where the seed variable has its impact. ICP automatically loads a file if the impact point refers to a position in some other file. With simple key stokes the user can browse through the impact chains in both forward and backward directions and can edit the programs simultaneously.

Chapter 4

Results and Discussion

4.1 Development Environment

The Power-Editor was developed under the **UNIX** environment. The assumptions taken for implementation are as follows:

1. Input programming language ANSI C. The grammar of the ANSI-C has been taken from Compiler Design in C [12] and has been modified.

2. The query language is SQL. The grammar is written by Leroy Cain taken from an web-achieve [5].

3. The impact analyzing tool analyses the program at the code level. It assumes the programs in ANSI-C with embedded ESQL statements. Therefore, the tool works for databases where programming in this combination is permitted. Oracle are examples of this kind.

Tools used for implementation are:

1. Lex for lexical analysis [16].

2. YACC for parsing [13].

3. The preprocessor of ANSI C complier.

4. The interfacing is made with the gnu Emacs editor. ICP is written in eLisp or emacs Lisp [17, 20] to talk to the editor.

4.1.1 Code Size

- Code length:

Tools	:	2683 loc	60858 bytes.
Intermediate Representation Generator	:	4163 loc	108665 bytes.
Impact Chain Generator	:	1952 loc	76304 bytes.
Interface with Editor	:	527 loc	11493 bytes.
Browser (eLisp code)	:	366 loc	8811 bytes.

- Tool Size (binaries):

Intermediate Representation Generator	:	352256 bytes.
Impact Analyzer+Interface	:	221184 bytes.
Browser (eLisp byte-code)	:	6467 bytes.

4.2 Test Results

The tools has been tested on few moderately large ANSI-C programs with considerable complexity. The test results are furnished in table 1. The test results are taken on DEC Alpha machine.

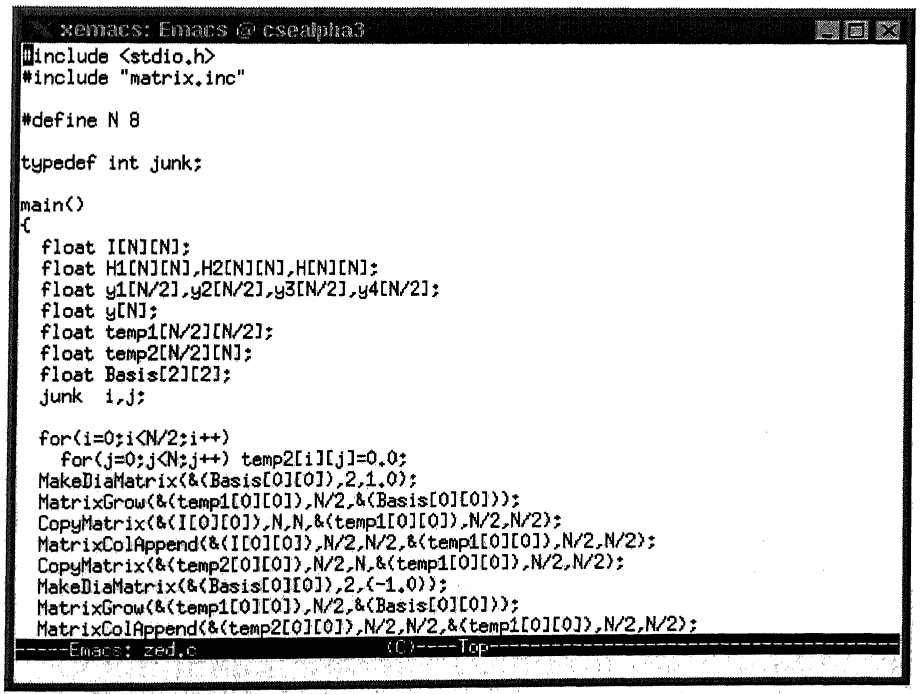
4.2.1 Capabilities of the Impact Analyzing Tool

- The tool identifies variables impacted by the seed variable. The seed variable can be any variable name, including a date-variable. In this sense it qualifies as a tool for the Year2000 problem. Otherwise it is a generalized tool.
- The tool assumes the input program to be correct.
- At the current stage, impact point identification through aliasing and pointers is not implemented.
- The IRG module of the tool requires further debugging for programs with embedded SQL statements.

Figure 7 through Figure 13 give snap-shots of the tool at work.
Appendix B shows some impact chains for different seed variables.

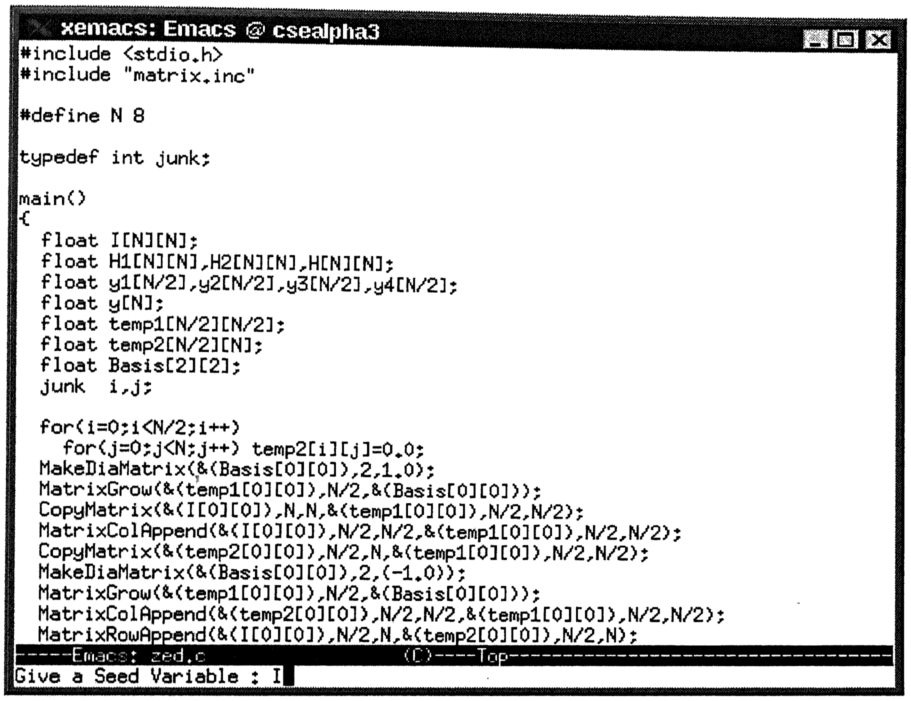
Program size (loc)	No. of files included	Chain length	CPU time (sec.)	Real Time (sec.)
2294	15	16	0.52	1.16
2294	15	37	0.65	1.21
2294	15	10	0.46	1.01
2294	15	18	0.45	1.18
2294	15	28	0.50	1.15
2618	17	5	0.25	0.88
2618	17	2	0.24	0.78
3166	18	25	0.39	1.02
3398	15	3	0.51	1.11
3398	15	7	0.49	1.10
3398	15	26	0.53	1.05
3398	15	9	0.47	1.09

Table 1: Test Results



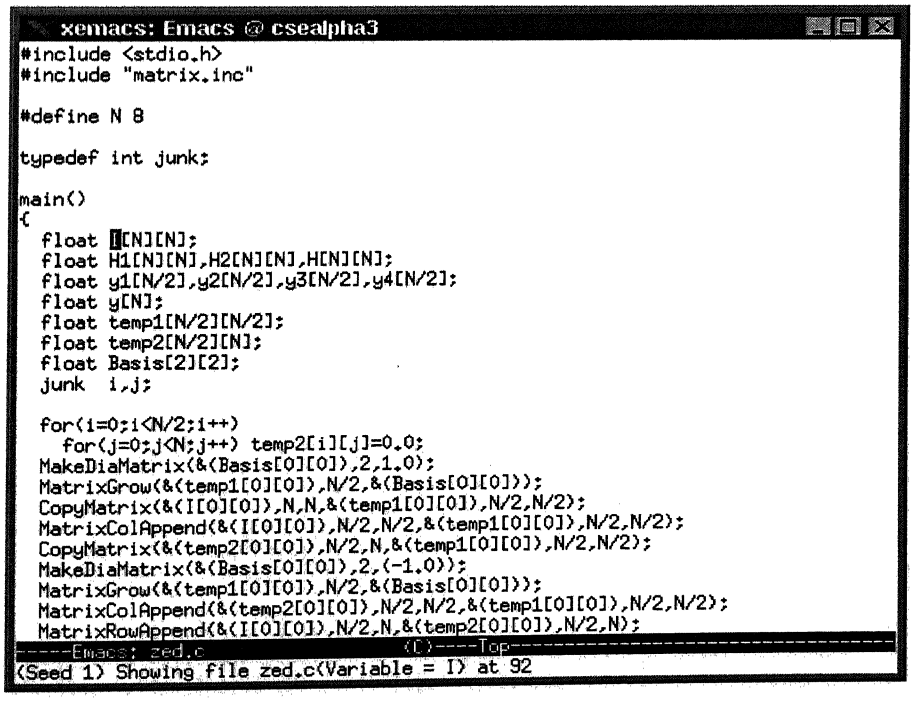
```
#include <stdio.h>
#include "matrix.inc"

#define N 8

typedef int junk;

main()
{
  float I[N][N];
  float H1[N][N],H2[N][N],H[N][N];
  float y1[N/2],y2[N/2],y3[N/2],y4[N/2];
  float y[N];
  float temp1[N/2][N/2];
  float temp2[N/2][N];
  float Basis[2][2];
  junk  i,j;

  for(i=0;i<N/2;i++)
    for(j=0;j<N;j++) temp2[i][j]=0.0;
  MakeDiaMatrix(&(Basis[0][0]),2,1.0);
  MatrixGrow(&(temp1[0][0]),N/2,&(Basis[0][0]));
  CopyMatrix(&(I[0][0]),N,N,&(temp1[0][0]),N/2,N/2);
  MatrixColAppend(&(I[0][0]),N/2,N/2,&(temp1[0][0]),N/2,N/2);
  CopyMatrix(&(temp2[0][0]),N/2,N,&(temp1[0][0]),N/2,N/2);
  MakeDiaMatrix(&(Basis[0][0]),2,(-1.0));
  MatrixGrow(&(temp1[0][0]),N/2,&(Basis[0][0]));
  MatrixColAppend(&(temp2[0][0]),N/2,N/2,&(temp1[0][0]),N/2,N/2);
-----Emacs: zed.c              (C)----Top-------------------------------
```

Figure 7: The main C file opened in gnu-emacs editor.



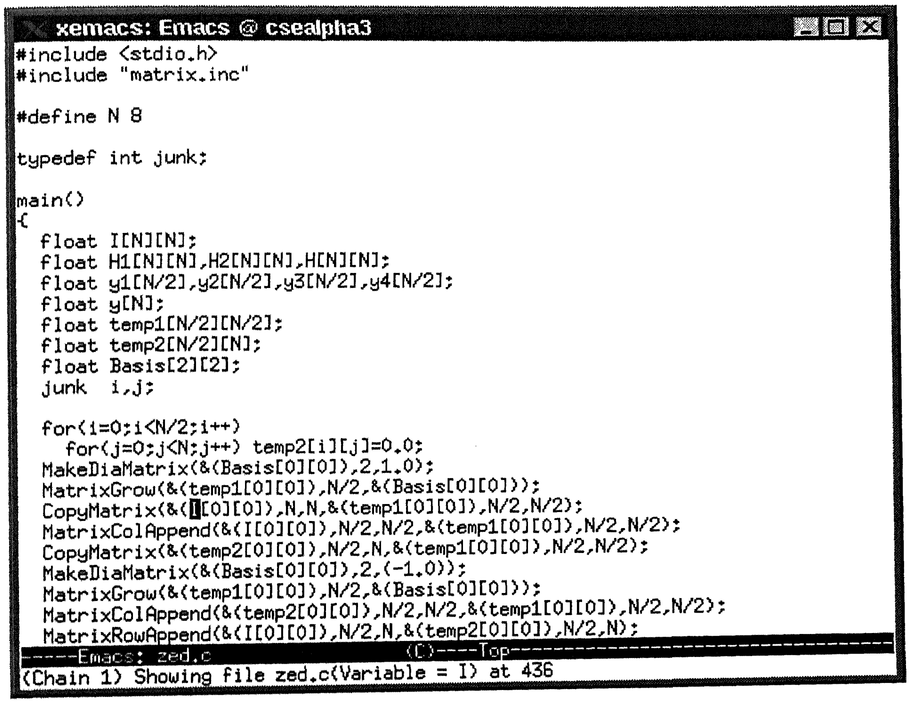
```
xemacs: Emacs @ csealpha3
#include <stdio.h>
#include "matrix.inc"

#define N 8

typedef int junk;

main()
{
  float I[N][N];
  float H1[N][N],H2[N][N],H[N][N];
  float y1[N/2],y2[N/2],y3[N/2],y4[N/2];
  float y[N];
  float temp1[N/2][N/2];
  float temp2[N/2][N];
  float Basis[2][2];
  junk  i,j;

  for(i=0;i<N/2;i++)
    for(j=0;j<N;j++) temp2[i][j]=0.0;
  MakeDiaMatrix(&(Basis[0][0]),2,1.0);
  MatrixGrow(&(temp1[0][0]),N/2,&(Basis[0][0]));
  CopyMatrix(&(I[0][0]),N,N,&(temp1[0][0]),N/2,N/2);
  MatrixColAppend(&(I[0][0]),N/2,N/2,&(temp1[0][0]),N/2,N/2);
  CopyMatrix(&(temp2[0][0]),N/2,N,&(temp1[0][0]),N/2,N/2);
  MakeDiaMatrix(&(Basis[0][0]),2,(-1.0));
  MatrixGrow(&(temp1[0][0]),N/2,&(Basis[0][0]));
  MatrixColAppend(&(temp2[0][0]),N/2,N/2,&(temp1[0][0]),N/2,N/2);
  MatrixRowAppend(&(I[0][0]),N/2,N,&(temp2[0][0]),N/2,N);
-----Emacs: zed.c                    (C)----Top-----------------------------
Give a Seed Variable : I
```

Figure 8: Input of Seed variable.



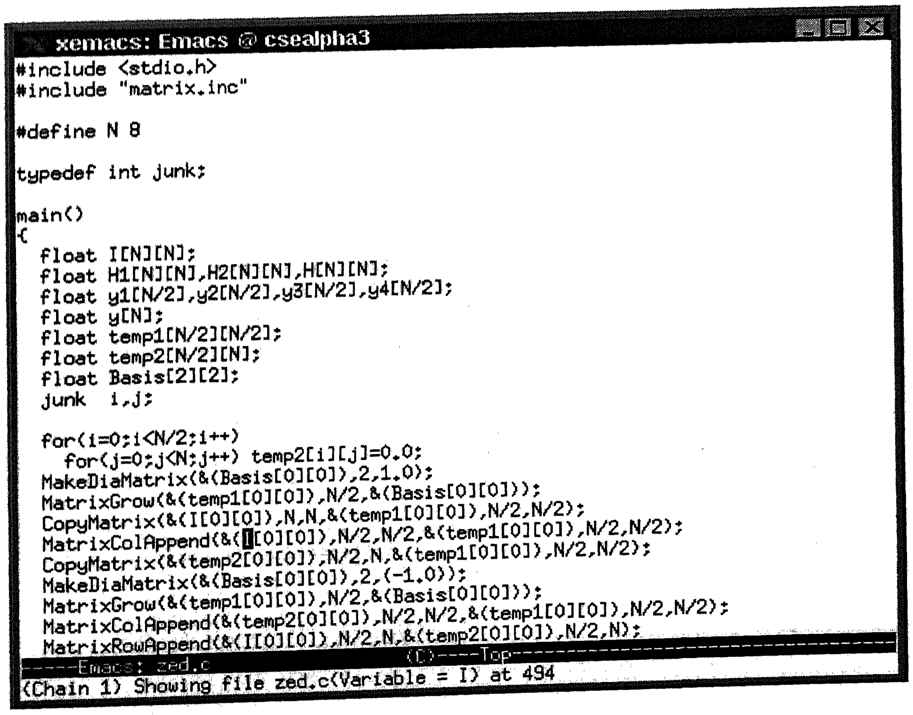
```
xemacs: Emacs @ csealpha3
#include <stdio.h>
#include "matrix.inc"

#define N 8

typedef int junk;

main()
{
  float I[N][N];
  float H1[N][N],H2[N][N],H[N][N];
  float y1[N/2],y2[N/2],y3[N/2],y4[N/2];
  float y[N];
  float temp1[N/2][N/2];
  float temp2[N/2][N];
  float Basis[2][2];
  junk  i,j;

  for(i=0;i<N/2;i++)
    for(j=0;j<N;j++) temp2[i][j]=0.0;
  MakeDiaMatrix(&(Basis[0][0]),2,1.0);
  MatrixGrow(&(temp1[0][0]),N/2,&(Basis[0][0]));
  CopyMatrix(&(I[0][0]),N,N,&(temp1[0][0]),N/2,N/2);
  MatrixColAppend(&(I[0][0]),N/2,N/2,&(temp1[0][0]),N/2,N/2);
  CopyMatrix(&(temp2[0][0]),N/2,N,&(temp1[0][0]),N/2,N/2);
  MakeDiaMatrix(&(Basis[0][0]),2,(-1.0));
  MatrixGrow(&(temp1[0][0]),N/2,&(Basis[0][0]));
  MatrixColAppend(&(temp2[0][0]),N/2,N/2,&(temp1[0][0]),N/2,N/2);
  MatrixRowAppend(&(I[0][0]),N/2,N,&(temp2[0][0]),N/2,N);
-----Emacs: zed.c                    (C)----Top-----------------------------
(Seed 1) Showing file zed.c(Variable = I) at 92
```

Figure 9: The first impact point.



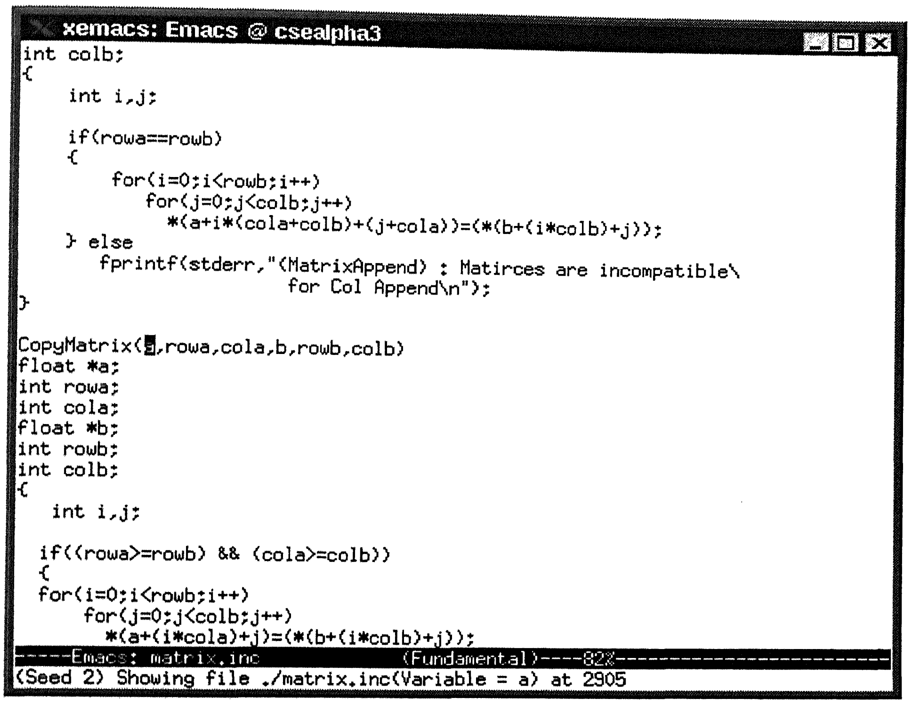
```
#include <stdio.h>
#include "matrix.inc"

#define N 8

typedef int junk;

main()
{
  float I[N][N];
  float H1[N][N],H2[N][N],H[N][N];
  float y1[N/2],y2[N/2],y3[N/2],y4[N/2];
  float y[N];
  float temp1[N/2][N/2];
  float temp2[N/2][N];
  float Basis[2][2];
  junk  i,j;

  for(i=0;i<N/2;i++)
    for(j=0;j<N;j++) temp2[i][j]=0.0;
  MakeDiaMatrix(&(Basis[0][0]),2,1.0);
  MatrixGrow(&(temp1[0][0]),N/2,&(Basis[0][0]));
  CopyMatrix(&(I[0][0]),N,N,&(temp1[0][0]),N/2,N/2);
  MatrixColAppend(&(I[0][0]),N/2,N/2,&(temp1[0][0]),N/2,N/2);
  CopyMatrix(&(temp2[0][0]),N/2,N,&(temp1[0][0]),N/2,N/2);
  MakeDiaMatrix(&(Basis[0][0]),2,(-1.0));
  MatrixGrow(&(temp1[0][0]),N/2,&(Basis[0][0]));
  MatrixColAppend(&(temp2[0][0]),N/2,N/2,&(temp1[0][0]),N/2,N/2);
  MatrixRowAppend(&(I[0][0]),N/2,N,&(temp2[0][0]),N/2,N);
-----Emacs: zed.c                   (C)----Top-----------------------
(Chain 1) Showing file zed.c(Variable = I) at 436
```

Figure 10: The second impact point.



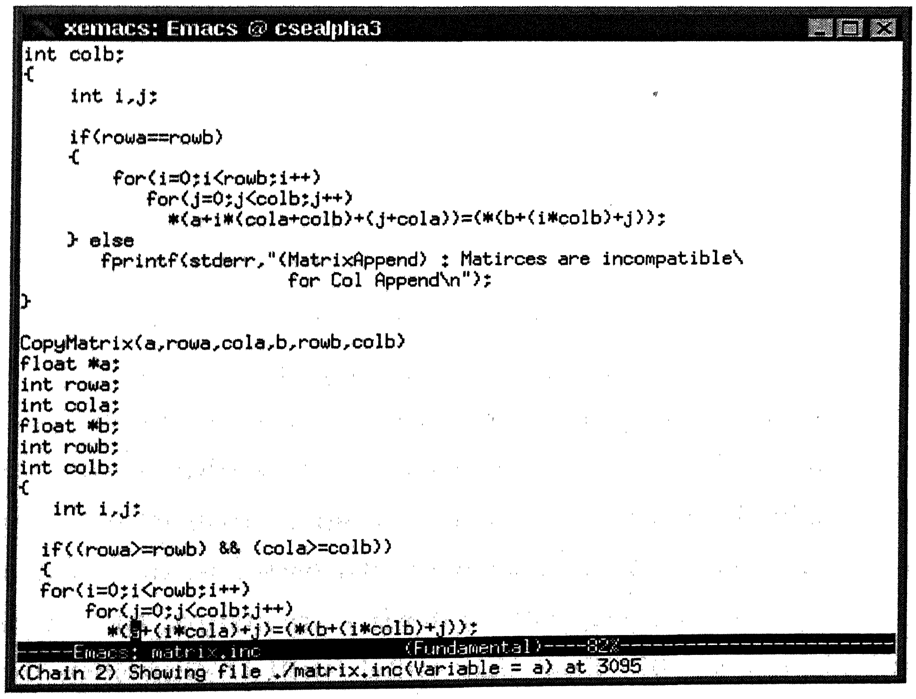
```
#include <stdio.h>
#include "matrix.inc"

#define N 8

typedef int junk;

main()
{
  float I[N][N];
  float H1[N][N],H2[N][N],H[N][N];
  float y1[N/2],y2[N/2],y3[N/2],y4[N/2];
  float y[N];
  float temp1[N/2][N/2];
  float temp2[N/2][N];
  float Basis[2][2];
  junk  i,j;

  for(i=0;i<N/2;i++)
    for(j=0;j<N;j++) temp2[i][j]=0.0;
  MakeDiaMatrix(&(Basis[0][0]),2,1.0);
  MatrixGrow(&(temp1[0][0]),N/2,&(Basis[0][0]));
  CopyMatrix(&(I[0][0]),N,N,&(temp1[0][0]),N/2,N/2);
  MatrixColAppend(&(I[0][0]),N/2,N/2,&(temp1[0][0]),N/2,N/2);
  CopyMatrix(&(temp2[0][0]),N/2,N,&(temp1[0][0]),N/2,N/2);
  MakeDiaMatrix(&(Basis[0][0]),2,(-1.0));
  MatrixGrow(&(temp1[0][0]),N/2,&(Basis[0][0]));
  MatrixColAppend(&(temp2[0][0]),N/2,N/2,&(temp1[0][0]),N/2,N/2);
  MatrixRowAppend(&(I[0][0]),N/2,N,&(temp2[0][0]),N/2,N);
-----Emacs: zed.c                   (C)----Top-----------------------
(Chain 1) Showing file zed.c(Variable = I) at 494
```

Figure 11: The third impact point.



```
int colb;
{
     int i,j;

     if(rowa==rowb)
     {
          for(i=0;i<rowb;i++)
              for(j=0;j<colb;j++)
                 *(a+i*(cola+colb)+(j+cola))=(*(b+(i*colb)+j));
     } else
         fprintf(stderr,"(MatrixAppend) : Matirces are incompatible\
                              for Col Append\n");
}

CopyMatrix(a,rowa,cola,b,rowb,colb)
float *a;
int rowa;
int cola;
float *b;
int rowb;
int colb;
{
    int i,j;

  if((rowa>=rowb) && (cola>=colb))
  {
  for(i=0;i<rowb;i++)
       for(j=0;j<colb;j++)
         *(a+(i*cola)+j)=(*(b+(i*colb)+j));
-----Emacs: matrix.inc              (Fundamental)----82%----------------------
(Seed 2) Showing file ./matrix.inc(Variable = a) at 2905
```

Figure 12: Variable impacted through function call.



```
int colb;
{
     int i,j;

     if(rowa==rowb)
     {
          for(i=0;i<rowb;i++)
              for(j=0;j<colb;j++)
                 *(a+i*(cola+colb)+(j+cola))=(*(b+(i*colb)+j));
     } else
         fprintf(stderr,"(MatrixAppend) : Matirces are incompatible\
                              for Col Append\n");
}

CopyMatrix(a,rowa,cola,b,rowb,colb)
float *a;
int rowa;
int cola;
float *b;
int rowb;
int colb;
{
    int i,j;

  if((rowa>=rowb) && (cola>=colb))
  {
  for(i=0;i<rowb;i++)
       for(j=0;j<colb;j++)
         *(a+(i*cola)+j)=(*(b+(i*colb)+j));
-----Emacs: matrix.inc              (Fundamental)----82%----------------------
(Chain 2) Showing file ./matrix.inc(Variable = a) at 3095
```

Figure 13: The next impact point in the function.

Chapter 5

Conclusion

In this thesis an analysis method has been developed for in-depth analysis of programs at the code level. This technique is called the *Impact Analysis* method. The analysis procedure is based on the data-flow analysis, a method commonly employed in the code optimization phase of the compilers. In the control-flow graph of a program, the technique traces the complete path of data/values, originating from a specified variable, called the *seed variable.* The data in its data-flow path may be associated with variables, other than the seed variable.

The impact analysis method has been fruitfully employed to make a better tool than the existing analysis-editing tools for the Year2000 problem. The existing tools do *address-based* analysis. They search programs syntactically for variables which have a pre-defined pattern and declare them as 'date-variables'. The other variables are 'non-date-variables'. Therefore, these tools are primarily pattern-matchers. The main weakness of 'address-based' analysis technique is that it fails to identify those non-date-variables, which during program execution may hold a date-value. In contrast to the 'address-based' analysis the impact analysis is a *value-based* analysis procedure. Impact analysis traces a 'value', which can be a date-value, and reveals the variables and the places in the program which is impacted by that value. Moreover, the analysis method systematically reduces the search space as it checks only those places in the program where the flow-of-control can reach.

We have implemented a Power-Editor which along with the normal editing facilities, has the capability to carry out impact analysis on program in the editor. Impact analysis can be carried out on programs written in ANSI-C with ESQL statements.

5.1 Future Work

At the present implementation the tool does not automatically find the date-variables from the program. The tool can be extended to include a pattern-matcher for automatic identification of date-variable. At the current state the tool cannot find impact points through aliasing and pointers and can be modified to take them into account.

Bibliography

[1] *The Year2000 and 2-digit Dates: A Guide for Planning and Implementation.* `http://www.software.ibm.com/year2000/resource.html`, October 1995.

[2] Aho, A. V., Shethi, R., and Ullman, J. D. *Compilers, Principles, Techniques and Tools.* Addison-Wesley, Reading, MA, 1986.

[3] Appleton, E. L. Call in the cavalry before 2000. *Datamation* (January 1996), pp.42–44.

[4] Baum, D. Tool up for 2000. *Datamation* (January 1996), pp.49–52.

[5] Cain, L., *SQL parser based on Lex/YACC.* `http://www.iiug.org/members/memb_software/archive/asql_yacc.`

[6] Celko, J. Start fixing Year2000 problems now! *Datamation* (January 1996), pp.36–38.

[7] de Jager, P., *Biting the silver bullet.* `http://www.year2000.com/y2karchive.html.`

[8] Department of Information Resources(DIR). *Standards, Review and Recommendations Publication, The Year 2000.* `http://www.itpolicy.gsa.gov.`

[9] Eldridge, A., and Louton, B. *A comparison of procedural and data change options for century compliance.* `http://www.year2000.com/y2karchieve.html.`

[10] Gothard, W., and Rodner, L. Strategies for solving the Y2K problem. *Dr. Dobb's Journal* (May 1998), pp.26–32.

[11] Hecht, M. *Flow Analysis of Computer Programs.* Elsevier North-Holland Inc., 1997.

[12] Holub, A. I. *Compiler Design in C.* Prentice Hall Software Series. Prentice Hall of India Pvt. Ltd., 1990.

[13] Johnson, S. C. *YACC - Yet Another Compiler Compiler.* Unix Programming Manual II, Bell Telephones Labs, Murray Hill, NJ, 1975.

[14] Kappelman, L. A. *Year 2000 Problem, Strategies and Solution from the Fortune 100*, 1st ed. Thompson Computer Press, December 1997.

[15] Leon, A., and Thomas, G. *IBM Mainframe and Year 2000 Solutions*, 4rth ed. Comdex Computer Publishing, November 1997.

[16] Lesk, M. E. *Lex - A lexical analyzer generator*, Computer Science Technical Report. Unix Programming Manual II, Bell Telephones Labs, Murray Hill, NJ, 1975.

[17] Lewis, B., LaLiberte, D., and the GNU Manual Group. *GNU Emacs Lisp Reference Manual, GNU Emacs version 18*, 1.03 ed. Free Software Foundation, December 1990.

[18] L.Moore, R., and Foley, D. G. Date compression and Year 2000 challenges. *Dr. Dobb's Journal* (May 1998), pp.20–23.

[19] Sandler, R. J. *Year 2000 Frequently Asked Questions*, 2.2 ed. `http://ourworld.compuserve.com/homepages/rsandler`, January 14 1997.

[20] Stallman, R. *GNU Emacs Manual, Emacs Version 18*, sixth ed. Free Software Foundation, February 1988.

[21] Yaeger, F. L. *DFSort* `ptf un90139` *Year 2000 and performance enhancement.* Tech. rep., DFSort team, IBM Storage Systems Division, `http://www.storage.ibm.com/software/soft.srtmhome.html`, April 1996.

[22] Zvegintzov, N. A resource guide to Year 2000 problem. *Computer* (March 1997), pp.58–64.

Appendix A

User's Manual for Power-Editor

A.1 About the Power-Editor

The Power-Editor is based on GNU-emacs editor. It provides extra functionality of doing *impact analysis* on the program in the current buffer of the editor. The user can invoke the *impact analyzing tool*(which does impact analysis) from the editor. When invoked the editor asks for a *seed variable* and brings out the *impacts* of the given variable. The user then can browse through the impact points using simple key-strokes and at the same time avail the normal editing functionalities of the editor.

A.2 How to use the Impact Analyzing Tool

Suppose the facility is available in the directory :

```
/usr/lib/emacs/lisp
```

then include the following lines in your `.emacs` file.

```
(setq load-path
      (append
         (list nil
               "/usr/lib/emacs/lisp"
```

```
                        load-path))
              )

        (require 'ped)
        (autoload 'ped "ped" nil t)

        (setq emacs-lisp-mode-hook
               (function (lambda ()
                      (require 'ped)
          (ped))
                     )
              )
```

Now, to invoke the tool the user have to load the file in the emacs editor and then invoke the function `Analysis` (`Meta-x Analysis`). The tool will ask for a seed variable. It then analyzes the program loaded in the emacs' current buffer and constructs the impact chain. Once the analysis is over the cursor shows the first impact point. The user then can edit the program. At this point the user can avail the facility of the following functions to browse through the impact points. Their functionalities are given along with. The user can as well use the editing facilities of the editor simultaneously.

`Next`	:	Takes the cursor to the next impact point.
`Prev`	:	Takes the cursor to the previous impact
	:	point.
`Next-Node`	:	It asks for a number and takes the cursor that many impact point forward.
`Prev-Node`	:	It asks for a number and takes the cursor that many impact point backwards.

To facilitate handling of these function, one can map them to some keys (as normally done with emacs). For example, here a set of key-maps definitions that binds the functions to the corresponding keys in the global key-map table.

```
(define-key global-map "\C-ca" 'Analysis)
(define-key global-map "\C-cn" 'Next)
(define-key global-map "\C-cp" 'Prev)
(define-key global-map "\C-cf" 'Next-Node)
(define-key global-map "\C-cb" 'Prev-Node)
```

if you include them in your `.emacs` file, you can get the following effects:

`Cntl+c a`	:	Asks for a seed variable and constructs the impact chain.
`Cntl+c n`	:	Takes the cursor to the next impact point.
`Cntl+c p`	:	Takes the cursor to the previous impact point.
`Cntl+c f`	:	Asks for a number and takes the that many impact point forward.
`Cntl+c b`	:	Asks for a number and takes the that many impact point backward.

Note: The impact analysis expects only correctly written programs in ANSI-C with embedded ESQL statements.

Appendix B

Test Data

B.1 Test Program

The impact analyzing tool has been tested on the following program. The program has an `included` file, which also has been listed below. The test results for 2 different variables has been given in the next section.

```
        /**********************************************/
        /*                                            */
        /*              File : zed.c (main file)      */
        /*                                            */
        /**********************************************/

#include <stdio.h>
#include "matrix.inc"

#define N 8

typedef int junk;

main()
{
```

```
float I[N][N];
float H1[N][N],H2[N][N],H[N][N];
float y1[N/2],y2[N/2],y3[N/2],y4[N/2];
float y[N];
float temp1[N/2][N/2];
float temp2[N/2][N];
float Basis[2][2];
junk  i,j;

for(i=0;i<N/2;i++)
  for(j=0;j<N;j++) temp2[i][j]=0.0;
MakeDiaMatrix(&(Basis[0][0]),2,1.0);
MatrixGrow(&(temp1[0][0]),N/2,&(Basis[0][0]));
CopyMatrix(&(I[0][0]),N,N,&(temp1[0][0]),N/2,N/2);
MatrixColAppend(&(I[0][0]),N/2,N/2,&(temp1[0][0]),N/2,N/2);
CopyMatrix(&(temp2[0][0]),N/2,N,&(temp1[0][0]),N/2,N/2);
MakeDiaMatrix(&(Basis[0][0]),2,(-1.0));
MatrixGrow(&(temp1[0][0]),N/2,&(Basis[0][0]));
MatrixColAppend(&(temp2[0][0]),N/2,N/2,&(temp1[0][0]),N/2,N/2);
MatrixRowAppend(&(I[0][0]),N/2,N,&(temp2[0][0]),N/2,N);
MatrixPrint(&(I[0][0]),N,N);


Basis[0][0]=1.0;
Basis[0][1]=1.0;
Basis[1][0]=1.0;
Basis[1][1]=(-1.0);
MatrixGrow(&(temp1[0][0]),N/2,&(Basis[0][0]));
CopyMatrix(&(H1[0][0]),N,N,&(temp1[0][0]),N/2,N/2);
MakeDiaMatrix(&(temp1[0][0]),N/2,(0.0));
MatrixColAppend(&(H1[0][0]),N/2,N/2,&(temp1[0][0]),N/2,N/2);
CopyMatrix(&(temp2[0][0]),N/2,N,&(temp1[0][0]),N/2,N/2);
```

```
  MatrixGrow(&(temp1[0][0]),N/2,&(Basis[0][0]));
  MatrixColAppend(&(temp2[0][0]),N/2,N/2,&(temp1[0][0]),N/2,N/2);
  MatrixRowAppend(&(H1[0][0]),N/2,N,&(temp2[0][0]),N/2,N);
  MatrixPrint(&(H1[0][0]),N,N);

  MatrixMult(&(I[0][0]),N,N,&(H1[0][0]),N,N,&(H[0][0]));
  CopyMatrix(&(H2[0][0]),N,N,&(H[0][0]),N,N);
  MatrixMult(&(H1[0][0]),N,N,&(H2[0][0]),N,N,&(H[0][0]));
  CopyMatrix(&(H2[0][0]),N,N,&(H[0][0]),N,N);
  MatrixPrint(&(H2[0][0]),N,N);

  printf("Enter the Y1 Matrix ----------\n");
  MatrixRead(&(y1[0]),N/2,1);
  printf("Enter the Y2 Matrix ----------\n");
  MatrixRead(&(y2[0]),N/2,1);
  printf("Enter the Y3 Matrix ----------\n");
  MatrixRead(&(y3[0]),N/2,1);
  printf("Enter the Y4 Matrix ----------\n");
  MatrixRead(&(y4[0]),N/2,1);

  MatrixSub(&(y1[0]),N/2,1,&(y2[0]),N/2,1);
  MatrixSub(&(y3[0]),N/2,1,&(y4[0]),N/2,1);
  CopyMatrix(&(y[0]),N,1,&(y1[0]),N/2,1);
  MatrixRowAppend(&(y[0]),N/2,1,&(y3[0]),N/2,1);

  MatrixMult(&(H2[0][0]),N,N,&(y[0]),N,1,&(H[0][0]));
  MatrixPrint(&(H[0][0]),N,1);

  for(i=0;i<N;i++)
    H[0][i]=H[0][i]/N;
  MatrixPrint(&(H[0][0]),N,1);
}
```

```
        /**********************************************/
        /*                                            */
        /*        File : matrix.inc (included file)   */
        /*                                            */
        /**********************************************/

#ifndef __MATRIX_INC__
#define __MATRIX_INC__

MatrixRead(a,row,col)
const float *a;
int row;
int col;
{
   int   i,j;
   float val;

   printf("\t");
   for(j=0;j<col;j++) printf("  %d\t",j+1);
   printf("\n");
   for(i=0;i<row;i++)
   {
      printf("Row %d\t",i+1);
      for(j=0;j<col;j++)
      {
  scanf("%f",&val);
  *(a+(i*col)+j)=val;
      }
      printf("\n");
   }
```

```
}


MatrixPrint(a,row,col)
float *a;
int row;
int col;
{
   int i,j;

   printf("\t");
   for(j=0;j<col;j++) printf("   %d\t",j+1);
   printf(" \n");
   for(i=0;i<row;i++)
   {
      printf(" %d\t",i+1);
      for(j=0;j<col;j++) printf(" %3.2f\t",*(a+(i*col)+j));
      printf("\n");
   }
}

MakeDiaMatrix(a,size,val)
float *a;
int size;
float val;
{
   int i,j;

   for(i=0;i<size;i++)
      for(j=0;j<size;j++)
*(a+(i*size)+j)=0;
   for(i=0;i<size;i++)  *(a+(i*size)+i)=val;
```

```
}

MatrixMult(a,rowa,cola,b,rowb,colb,c)
float *a;
int rowa;
int cola;
float *b;
int rowb;
int colb;
float *c;
{ int i,j,k;

   if(cola==rowb)
   {
       for(i=0;i<rowa;i++)
  for(j=0;j<colb;j++)
      *(c+(cola*i)+j)=0;
       for(i=0;i<rowa;i++)
  for(j=0;j<cola;j++)
    for(k=0;k<colb;k++)
 *(c+(i*colb)+k)+=((*(a+(i*cola)+j))*(*(b+(j*colb)+k)));
   } else
      fprintf(stderr,"(MatMult)  : Matrices are incompatible for \
Matrix Multiplication\n");
}

MatrixSub(a,rowa,cola,b,rowb,colb)
float *a;
int rowa;
int cola;
float *b;
int rowb;
int colb;
{
   int i,j;

   if((rowa==rowb) &&(cola==colb))
   {
       for(i=0;i<rowa;i++)
          for(j=0;j<colb;j++)
      *(a+(i*cola)+j)-=(*(b+(colb+i)+j));
   } else
fprintf(stderr,"(MatSub) : Matrices are incompatible for\
Matrix subtraction\n");
}

MatrixGrow(a,size,Basis)
float *a;
int   size;
float *Basis;
{
    int   row,col;
    int   i,j;

    *(a)=(*(Basis));
    *(a+1)=(*(Basis+1));
    *(a+size)=(*(Basis+2));
    *(a+size+1)=(*(Basis+3));
    row=col=2;

    while(row<size)
    {
      for(i=0;i<row;i++)
  for(j=0;j<col;j++)
```

```
 {
     *(a+(i*size)+(j+col))         = (*(a+(i*size)+j)) * (*(Basis+1));
     *(a+((i+row)*size)+j)         = (*(a+(i*size)+j)) * (*(Basis+2));
     *(a+((i+row)*size)+(j+col)) = (*(a+(i*size)+j)) * (*(Basis+3));
 }
       row*=2; col*=2;
    }
}

MatrixRowAppend(a,rowa,cola,b,rowb,colb)
float *a;
int rowa;
int cola;
float *b;
int rowb;
int colb;
{
   int i,j;

   if(cola==colb)
   {
       for(i=0;i<rowb;i++)
 for(j=0;j<colb;j++)
    *(a+((i+rowa)*cola)+j)=(*(b+(i*colb)+j));
   } else
       fprintf(stderr,"(MatrixAppend) : Matirces are incompatible\
                                    for Row Append\n");


}

MatrixColAppend(a,rowa,cola,b,rowb,colb)
float *a;
```

```
int rowa;
int cola;
float *b;
int rowb;
int colb;
{
    int i,j;

    if(rowa==rowb)
    {
for(i=0;i<rowb;i++)
   for(j=0;j<colb;j++)
     *(a+i*(cola+colb)+(j+cola))=(*(b+(i*colb)+j));
    } else
       fprintf(stderr,"(MatrixAppend) : Matirces are incompatible\
for Col Append\n");
}

CopyMatrix(a,rowa,cola,b,rowb,colb)
float *a;
int rowa;
int cola;
float *b;
int rowb;
int colb;
{
   int i,j;

  if((rowa>=rowb) && (cola>=colb))
  {
  for(i=0;i<rowb;i++)
      for(j=0;j<colb;j++)
*(a+(i*cola)+j)=(*(b+(i*colb)+j));
  } else
       fprintf(stderr,"(CopyMatrix) : Matrices Incompatible for Copying
}

#endif
```

B.2 Impact Chains

The impact chains are a series of *impact points*. Each impact points are represented according to the following format:

< "File Name" , Variable Name (Row No. , Column No.)

The *seed definitions* are labeled as 'Seed :' along the left margin with its *uses*, which follows it.

```
Impact Analysis for
Seed Variable = I
---------------------


Seed : <"zed.c",I(10,8)>
          <"zed.c",I(23,15)>
          <"zed.c",I(24,20)>
          <"zed.c",I(29,20)>
          <"zed.c",I(30,16)>
          <"zed.c",I(47,15)>
Seed : <"./matrix.inc",a(170,11)>
          <"./matrix.inc",a(184,3)>
Seed : <"./matrix.inc",a(150,16)>
```

```
          <"./matrix.inc",a(164,8)>
Seed : <"./matrix.inc",a(130,16)>
          <"./matrix.inc",a(144,7)>
Seed : <"./matrix.inc",a(28,12)>
          <"./matrix.inc",a(41,45)>
Seed : <"./matrix.inc",a(59,11)>
          <"./matrix.inc",a(77,24)>


Impact Analysis for
Seed Variable = Basis
------------------------


Seed : <"./matrix.inc",Basis(103,18)>
          <"./matrix.inc",Basis(122,59)>
          <"./matrix.inc",Basis(123,59)>
          <"./matrix.inc",Basis(124,59)>
Seed : <"./matrix.inc",Basis(106,7)>
          <"./matrix.inc",Basis(111,12)>
          <"./matrix.inc",Basis(112,14)>
          <"./matrix.inc",Basis(113,17)>
          <"./matrix.inc",Basis(114,19)>
          <"./matrix.inc",Basis(122,59)>
          <"./matrix.inc",Basis(123,59)>
          <"./matrix.inc",Basis(124,59)>
Seed : <"zed.c",Basis(16,8)>
          <"zed.c",Basis(21,18)>
          <"zed.c",Basis(22,34)>
          <"zed.c",Basis(26,18)>
          <"zed.c",Basis(27,34)>
          <"zed.c",Basis(42,34)>
Seed : <"zed.c",Basis(33,2)>
```

```
          <"zed.c",Basis(42,34)>
Seed : <"zed.c",Basis(34,2)>
          <"zed.c",Basis(42,34)>
Seed : <"zed.c",Basis(35,2)>
          <"zed.c",Basis(42,34)>
Seed : <"zed.c",Basis(36,2)>
          <"zed.c",Basis(37,34)>
          <"zed.c",Basis(42,34)>
Seed : <"./matrix.inc",a(46,14)>
          <"./matrix.inc",a(55,3)>
          <"./matrix.inc",a(56,26)>
Seed : <"./matrix.inc",a(103,11)>
          <"./matrix.inc",a(122,8)>
          <"./matrix.inc",a(122,39)>
          <"./matrix.inc",a(123,8)>
          <"./matrix.inc",a(123,39)>
          <"./matrix.inc",a(124,8)>
          <"./matrix.inc",a(124,39)>
```